# From Events to Reactions: A Progress Report

Tony Garnock-Jones
Northeastern University, Boston, Massachusetts, USA
tonyg@ccs.neu.edu

SYNDICATE is a new coordinated, concurrent programming language. It occupies a novel point on the spectrum between the shared-everything paradigm of threads and the shared-nothing approach of actors. SYNDICATE actors exchange messages and share common knowledge via a carefully controlled database that clearly scopes conversations. This approach clearly simplifies coordination of concurrent activities. Experience in programming with SYNDICATE, however, suggests a need to raise the level of linguistic abstraction. In addition to writing event handlers and managing event subscriptions directly, the language will have to support a reactive style of programming. This paper presents event-oriented SYNDICATE programming and then describes a preliminary design for augmenting it with new reactive programming constructs.

## 1 Introduction

Interactive programs must keep track of their conversations with entities in the outside world, demultiplexing incoming events and maintaining internal state associated with each concurrent conversation. For example, web servers manage multiple conversations with browsers that come and go unpredictably. Even sequential programming languages face the challenges of this form of concurrency.

SYNDICATE [9, 10] aims to ease management of a program's interactions with the outside world. The language is centred around functional actors which exchange events and actions with their context. The Actor model [13] gives each actor a piece of private state, eschews shared state, and offers point-to-point delivery of messages to peers in a single, flat group. SYNDICATE loosens these restrictions with three novel features: *multicast* message delivery instead of point-to-point messaging;[1] a carefully controlled *shared dataspace* allowing peers to collaboratively maintain shared state; and *hierarchical containment* allowing decomposition of one actor into many, able to communicate internally via a private message bus and group-private shared dataspace.

An actor in SYNDICATE is either a *leaf actor*, with behaviour given by a programmer-supplied function, or a *network actor*, with behaviour specified by the language itself. A network actor contains and manages other actors, providing shared message bus and dataspace services to them. Messages are routed to actors based on pattern-matching *subscriptions* rather than actor IDs.

The active subscriptions in a network are just one kind of shared state. In the actor model, the (implicit) routing table connecting peers is the *only* shared state.[2] SYNDICATE generalises the routing table and allows actors to place arbitrary data, known as *assertions*, within it. The resulting structure is dubbed a *dataspace*. Subscriptions—assertions of *interest* in data of a particular shape—serve not only to route messages toward an actor, but to match assertions in the dataspace. Networks notify interested contained actors of the existence of assertions relevant to their interests, sending *state change notification* events as

[1] More properly, multicast delivery *subsumes* point-to-point messaging: the latter can be encoded using the former.

[2] While the *model* enforces isolation, practical actor *languages* almost always include ad-hoc mechanisms for sharing state. Often this is not an explicit design decision but a reflection of the impracticality of removing or controlling existing shared-state facilities in an underlying language such as Java, Javascript or C++.

Submitted to:
PLACES 2016

```
;;;; Actions available to actors, and events delivered to actors:
(spawn <function-expr> <state-expr> <actions-expr>)
                              ;; Action that spawns a new leaf actor.
(message <body-expr>)         ;; A "message" event/action.
(patch <added-assertions> <removed-assertions>)
                              ;; A "patch" state change notification event/action. Contains
                              ;; two sets: assertions added and removed.

;;;; Routines for building and composing patches:
(assert <expr>)               ;; Build patches that, when interpreted, add/remove assertions
(retract <expr>)              ;; to/from an actor's assertion set.
(patch-seq <expr> ...)        ;; Collapses patches into an equivalent cumulative patch.

;;;; Results available to be returned from behaviour functions:
(transition <state-expr> <actions-expr>)
                              ;; Tuple of updated actor state plus a sequence of actions,
                              ;; from a leaf actor's behaviour function.
(quit <actions-expr>)         ;; If returned in place of a transition, causes clean
                              ;; termination after performing some final actions.

;;;; Primitives for assertion sets and assertions of interest:
?                             ;; Wildcard value, representing the (infinite) set of
                              ;; all possible assertions.
(observe <assertion-expr>)    ;; Assertion of interest in the contained assertions.
```

Figure 1: Selected SYNDICATE constructs as expressed in Racket

assertions come and go and as actors assert new and retract old subscriptions. In this way, dataspaces are similar to Linda's tuplespaces [11]. However, unlike the "generative" tuples of Linda, assertions in SYNDICATE do not have independent lifetimes. Instead, an assertion remains in a network's dataspace only so long as some contained actor continues to assert it. When actors terminate, cleanly or abnormally, all their assertions are removed, and interested parties are notified of the change. This yields a failure signalling mechanism that is part of SYNDICATE's approach to lifecycle management of components.

Networks deliver events to their actors, which reply with actions to be interpreted by the network. Messages are sent and new actors created roughly as in the traditional actor model, but assertions, including subscriptions, are managed by *state change notifications*. An actor sends such notifications to its containing network to manipulate the set of active assertions associated with the actor and shared with the group. Symmetrically, networks send such notifications to contained actors to keep them up-to-date as assertions relevant to their interests come and go. State change notifications take the form of *patches* against a previous assertion set, describing assertions to be added to and removed from the set.

SYNDICATE programmers write behaviour functions that react to incoming events and yield updated actor state and a sequence of actions to perform. However, directly programming to this basic interface quickly becomes inconvenient. To fully achieve its goal of easing the management of concurrent interactions between a program and its environment, SYNDICATE needs higher-level constructs.

## 2 SYNDICATE By Example

The primary implementation of SYNDICATE is an embedded language based on Racket [7]. The language provides data structures and syntactic constructs corresponding to SYNDICATE's mathematical definition [9]. Some of these are shown in Figure 1. Events and actions are represented by Racket `structs`,

```
(define (manager e balance)
  (match-event e
    [(message (deposit amount))
     (transition (+ balance amount)
                 (patch-seq (retract (account balance))
                            (assert (account (+ balance amount)))))]))

(define (observer e _)
  (when (patch? e)
    (for [(balance (project-assertions (patch-added e) (account (?!))))]
      (printf "Balance changed to ~a\n" balance))))

(define (updater e _)
  (when (patch/added? e)
    (quit (list (message (deposit +100))
                (message (deposit -30))))))
```

Figure 2: Bank account actor behaviour functions

and behaviour functions associated with leaf actors are represented by Racket functions.

Programming in SYNDICATE involves the design of *protocols* specifying message types, interaction patterns, and roles that protocol participants must play. In addition, unlike protocols for actor systems, SYNDICATE protocols must specify data to be placed in the shared dataspace along with instructions for its maintenance. Each network actor thus embodies a protocol instance, and its contained actors are the protocol participants.

As is traditional, our example is a drastically oversimplified simulated bank account. The protocol involves three roles: account manager, account updater, and account observer. Only one account manager is permitted per protocol instance, but any number of updaters and observers may exist. The account manager maintains an `account` structure in the shared dataspace to describe the account balance. Account observers monitor this structure, tracking changes in the balance. Account updaters send `deposit` messages, which are interpreted by the account manager. The structures concerned are defined as follows:

```
(struct account (balance))  ;; Assertion: the current account balance is 'balance'
(struct deposit (amount))   ;; Message: instruction to add 'amount' to the balance
```

The role of account manager is performed by the leaf actor created by interpretation of the following *spawn* action specifying a behaviour function, an initial state, and a collection of startup actions:

```
(spawn manager 0 (patch-seq (assert (account 0)) (assert (observe (deposit ?)))))
```

The new actor's state is the account balance, initially zero. Its sole startup action is a state change notification—a patch—asserting two facts: that the current balance (in an `account` structure) is zero, and that the actor is interested in receiving `deposit` messages. Its behaviour function (`manager`, from Figure 2) lets it react to such messages: after computing an updated balance, it yields its new state (namely, the new balance) along with a patch action updating the contents of the shared dataspace with the new value. The patch is computed as the cumulative effect of two simpler patches: the first retracts the old assertion, and the second asserts the new.

An observer that prints the new balance each time it changes is created as follows:

```
(spawn observer (void) (assert (observe (account ?))))
```

The observer is locally stateless: its state throughout its lifetime is `(void)`. Its initial action asserts its interest in assertions of shape `(account ?)`. In response, its containing network—the implicit *ground*

*network* at the outermost layer of the running actor hierarchy—sends state change notification events keeping it informed of the `account` assertions in the dataspace. The observer responds to each such event by *projecting out* and printing the `balance` field of each newly-added `account` assertion. To do this, it uses a projection specification including a *capture* operator, written `(?!)`. The specification and the set of newly-added assertions are given to the function `project-assertions`. The result is a Racket set containing the captured values, if any. The observer loops over the set's contents, printing each value.

Finally, we exercise the account via an actor performing the role of an account updater:

```
(spawn updater (void) (assert (observe (observe (deposit ?)))))
```

It would be disastrous to send `deposit` messages before the account manager was listening for them. The updater therefore delays sending until interest in the messages to be sent is detected in the shared dataspace. Its initial patch, then, asserts interest in *assertions of interest in* `deposit` *messages*. Once its behaviour function receives a patch containing a non-empty added set, it can conclude that some peer is ready to receive `deposit` messages. It then sends two such messages before terminating cleanly.

The result of all this is that the account's balance, reflected as `account` assertions in the shared dataspace, starts at zero, climbs to 100, and finally drops to 70. No further activity occurs.

## 3 Reactive Constructs for SYNDICATE

The example, simple though it is, highlights several SYNDICATE idioms deserving of linguistic support. The preliminary "reactive" constructs presented in this section aim to capture the essence of such idioms:

**Input.** Actors wish to receive messages, but must manage assertions of interest in messages separately from the code responding to each message.

Similarly, actors wish to track the contents of the shared dataspace, and must not only manage assertions of interest in the relevant portion of the dataspace but also apply projections to the sets of assertions they are informed about.

**Output.** Actors not only send messages, but maintain sets of assertions relevant to peers.

**Demultiplexing.** Actors engage in multiple conversations at once, each handling one facet of the actor's overall interface to the outside world. Occasionally, the assertions of one ongoing conversation will *overlap* with those of another. The actor must take care not to inadvertently affect other conversations when adjusting one conversation's assertion set. Also, the actor must carefully examine each incoming event to determine which of its active conversations it might relate to.

**Auxiliary conversations.** Events can trigger complex chains of actions. For example, as part of reacting to an event, an actor may wish to call some other actor in the system, sending a message and continuing where it left off upon receiving a reply. It must then keep track of waiting continuations.

Figure 3 shows the proposed additions to the language. In the functional interface to SYNDICATE, actors manipulate *action structures* describing intended actions as values. By contrast, the new `send!` form is imperative, sending a message as a side effect. Likewise, `actor` creates, as a side effect, a new actor that executes the contained expressions sequentially.

At the heart of the new design is the `state` form and the related concept of *facets*. A `state` form is a *blocking* construct that stays active until one of its *termination events* occurs. When this happens, the expressions associated with the particular termination event are evaluated and the resulting values returned to the waiting continuation of the `state`. The `until` and `forever` forms are special cases of `state` with one and zero termination events, respectively.

```
;; Expressions added to the Racket language:
(actor <expr> ...)                (state [⟨#:collect [(<id> <expr>) ...]⟩ <facet> ...]
(send! <expr>)                      [<event> <expr> ...]
                                    ...)
                                  (until <event> ⟨#:collect [(<id> <expr>) ...]⟩ <facet> ...)
;; Events:                        (forever ⟨#:collect [(<id> <expr>) ...]⟩ <facet> ...)
(message <pattern>)
(asserted <pattern>)              ;; Facets:
(retracted <pattern>)             (assert <assertion-expr>)
(rising-edge <expr>)              (on <event> <expr> ...)
```

Figure 3: "Reactive" SYNDICATE constructs. $\langle\cdots\rangle$ indicates optional syntax.

While a `state` is running, its associated facets are active. A facet is either an `assert`, which keeps an assertion set in the shared dataspace up-to-date as its actor's state evolves, or an `on`, which responds to incoming events. If `#:collect` is present in a `state` form, the bindings next to it are in scope in all of the `state`'s facets. Each `on` facet's body must return as many values as there are `#:collect`ed bindings, making the entire `state` form into a kind of fold over events.

There are four kinds of events that can trigger evaluation of an `on` facet's body or one of the termination conditions of a `state`. A `message` event entails a subscription to the associated message pattern, and is triggered when a matching message comes in. An `asserted` event is fired for *each* assertion added to the shared dataspace and matching the given pattern. A `retracted` event is similar, but for assertions removed from the dataspace. Patterns in these event specifications use "$" to introduce binders, and "_" as a wildcard. Finally, `rising-edge` events trigger whenever, after some event arrives from the outside world, the given Racket expression evaluates to a true value, having previously evaluated to false.

The following example demonstrates the basic ideas:

```
(actor (send! 'starting)
       (define final-count
         (state [#:collect [(count 0)]                                  ;; (a,b)
                 (assert (list 'incrs-seen-so-far count))               ;; (b)
                 (on (message 'incr) (+ count 1))]                      ;; (a)
           [(rising-edge (>= count 5)) (send! 'too-many)    count]      ;; (c)
           [(message 'interrupt)       (send! 'interrupted) count])) ;; (c)
       (send! 'finished))
```

The actor shown sends a message, `'starting`, and then enters a `state` which (a) counts `'incr` messages, (b) maintains an assertion in the shared dataspace describing the count seen so far, and (c) waits either for the count to reach 5 or for an `'interrupt` message. Once one of the latter events happens, a message is sent before the final count is returned to the continuation of the `state`. Once the state terminates, the `'incrs-seen-so-far` assertion is retracted along with the event handlers monitoring `'incr` messages and the termination conditions. The actor then terminates after sending a `'finished` message.

Together, the constructs of Figure 3 support the idioms listed in the introduction to this section. Each `on` facet is automatically translated into code for maintaining appropriate assertions of interest, depending on the kind of event acting as its trigger. Facets awaiting assertions and retractions automatically project incoming patches and iterate over the resulting sets. Each `assert` facet automatically reevaluates its associated expression and, when it changes, updates the shared dataspace. Run-time support routines multiplex the assertions of the active facets, ensuring that they do not interfere, and dispatch incoming events to relevant facets. Finally, the blocking nature of `state` expressions lets event handler code suspend until some condition obtains. Unlike Erlang's selective receive construct, the actor's active facets remain

responsive while a sequence of expressions is suspended waiting for termination of a `state`. In other words, ongoing conversations are not disrupted by the temporary existence of an auxiliary conversation. An analogous distinction can be seen between modal and non-modal dialogs in graphical user interfaces.

**Revisiting the bank account example.** The new constructs shorten our example and clarify the purpose of the code. The account manager is expressed as follows:

```
(actor (forever #:collect [(balance 0)]
                (assert (account balance))
                (on (message (deposit $amount))
                    (+ balance amount))))
```

The account observer becomes:

```
(actor (forever (on (asserted (account $balance))
                    (printf "Balance changed to ~a\n" balance))))
```

Finally, our updater is simply:

```
(until (asserted (observe (deposit _))))
(send! (deposit +100))
(send! (deposit -30))
```

The `until` expression blocks waiting for an `(observe (deposit _))` assertion to appear in the shared dataspace. The `(on (message (deposit $amount)) ...)` facet in the account manager automatically establishes just such an assertion, thereby indirectly releasing the balance-alteration messages, now known to have a recipient.

**Implementation.** The prototype implementation is a library of macros and support routines that may be used to write any given leaf actor. The support routines use Racket's delimited continuations [8] to provide a direct-style, imperative veneer atop the functional interface between a leaf actor and its containing network demanded by SYNDICATE.

## 4 Related Work

SYNDICATE's dataspaces must be discussed in terms of comparison not only to other *models* of concurrent computation but also to practical actor *languages*. At the model level, dataspaces are related to Linda's *tuplespaces* [11], and its state change notifications to Erlang's failure-signalling mechanisms, *links* and *monitors* [2]. Tuples in Linda's tuplespaces are "generative"; that is, once created, they take on independent existence. SYNDICATE's assertions, however, survive only as long as some actor continues to assert them. The resulting failure-signalling facility is comparable to Erlang's links and monitors, which allow actors to subscribe to messages signalling the termination of specified peers. Cast in SYNDICATE terms, if an actor $X$ asserts its existence, `(ok` $X$`)`, then monitoring peers can react to `(retracted (ok` $X$`))`.

At the language level, dataspaces are quite different to forms of general-purpose shared state seen in actor languages. SYNDICATE actors can only affect a dataspace by placing assertions into it. In exchange for this severe restriction, automatic retraction of assertions on actor termination protects the structure from damage. By contrast, Erlang's ETS tables are effectively read/write hash tables shared among processes. They suffer many of the same problems of other kinds of shared, mutable storage, including no reliable way to detect or repair damage after abnormal exit of a process. Languages such as Akka, working within systems making liberal use of mutable, shared state, suggest as a matter of convention the use of

immutable data structures, but cannot enforce this. The resulting shared state facility is orthogonal to the language's actor model. It is worth noting that our Racket and Javascript SYNDICATE implementations also go beyond the SYNDICATE model: they suggest, but do not enforce, restrictions on use of mutable state. Finally, mention must be made of the language Pony, which employs *deny capabilities* [3] to efficiently and safely share access to objects. While SYNDICATE lacks shared memory of this kind, it could be used as a substrate for Pony's distributed resource management protocols, or Pony could be used to implement a single SYNDICATE leaf actor as a group of collaborating Pony actors.

Turning to the reactive constructs described in this paper, Hancock's language "Flogo II" [12] embodies ideas that are in some ways similar. Hancock introduces a "step/process distinction" and aims for a "unified paradigm" midway between "declarative" and "procedural" approaches to interactive programming. "Steps" in Flogo II are analogous to imperative actions in Racket, and "processes" are roughly analogous to facets, SYNDICATE's ongoing reactive constructs within blocking `state` expressions. SYNDICATE is clearly aiming at similar territory to Flogo II; however, the treatment of private and shared state in the Flogo II prototype is quite different, and the language lacks a full treatment of concurrency, message passing and error recovery. The reactive constructs presented here, by contrast, rest upon SYNDICATE's event-oriented formal semantics.

The term "reactive" is used in many contexts. It is used here to connote similarity to languages such as Dedalus [1] and to models such as functional reactive programming [5, 4] and even spreadsheets. Dedalus programs are structured around a distributed Datalog variant similar to SYNDICATE's dataspace. They react to changes in the Datalog database in some ways like SYNDICATE actors reacting to changes in the shared dataspace. Functional reactive programs construct dataflow graphs atop streams of time-varying values; if we view the time-varying set of assertions matching some expressed interest as a representation of such a stream, we can imagine a SYNDICATE actor implementing the computation at a dataflow graph node and publishing its result downstream as an assertion in turn. An entire program would then be a collection of such actors. Finally, the spreadsheet model, in which cells subscribe to updates from their peers and publish their own contents in turn, can be directly encoded in SYNDICATE. It remains future work to make these connections to other reactive designs precise.

## 5 Conclusion

This paper has given a brief overview of event-oriented programming in SYNDICATE, and motivated and introduced a preliminary design for a "reactive" style of programming in SYNDICATE. The design captures several important idioms of SYNDICATE programming, and makes programs employing these idioms both shorter and easier to read and understand.

The presented design is not only preliminary, but partial: ongoing work includes the exploration of constructs for automating the integration of incoming state change notifications to yield various aggregate structures and to compute predicates over the dataspace. In addition, the `state` form always creates a new actor, even when used in tail position. A future refinement should lift this limitation. Finally, future work will also include formal specification of the mapping from the proposed constructs to SYNDICATE's core semantics.

**Acknowledgements.** This work was supported in part by several NSF grants. The author would like to thank the anonymous reviewers for their feedback, comments and suggestions. In addition, many thanks to Matthias Felleisen, Sam Caldwell, and the participants of NU PLTs coffee round.

**Resources.** The programs shown in this paper, along with other examples and resources, may be found via `www.ccs.neu.edu/home/tonyg/places2016/`.

## References


[1] Peter Alvaro, William Marczak, Neil Conway, Joseph M. Hellerstein, David Maier & Russell C. Sears (2009): *Dedalus: Datalog in Time and Space*. Technical Report UCB/EECS-2009-173, EECS Department, University of California, Berkeley, doi:10.1007/978-3-642-24206-9_16.

[2] Joe Armstrong (2003): *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm.

[3] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing & Andy McNeil (2015): *Deny capabilities for safe, fast actors*. In: *Proc. Intl. Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE!*, doi:10.1145/2824815.2824816.

[4] Gregory H. Cooper & Shriram Krishnamurthi (2006): *Embedding dynamic dataflow in a call-by-value language*. In: *ESOP*, doi:10.1007/11693024_20.

[5] Conal Elliott & Paul Hudak (1997): *Functional reactive animation*. In: *ICFP*, doi:10.1145/258948.258973.

[6] Ericsson(AB) (2012): *Erlang/OTP Design Principles*. `http://www.erlang.org/doc/design_principles/des_princ.html`.

[7] Matthew Flatt & PLT (2010): *Reference: Racket*. Technical Report PLT-TR-2010-1, PLT Inc. Available at `http://racket-lang.org/tr1/`.

[8] Matthew Flatt, Gang Yu, Robert Bruce Findler & Matthias Felleisen (2007): *Adding delimited and composable control to a production programming environment*. In: *ICFP*, pp. 165–176, doi:10.1145/1291220.1291178.

[9] Tony Garnock-Jones & Matthias Felleisen (2016): *Coordinated Concurrent Programming in Syndicate*. In: *ESOP*. To appear.

[10] Tony Garnock-Jones, Sam Tobin-Hochstadt & Matthias Felleisen (2014): *The Network as a Language Construct*. In: *ESOP*, pp. 473–492, doi:10.1007/978-3-642-54833-8_25.

[11] David Gelernter (1985): *Generative communication in Linda*. *TOPLAS* 7(1), pp. 80–112, doi:10.1145/2363.2433.

[12] Christopher Michael Hancock (2003): *Real-time programming and the big ideas of computational literacy*. Ph.D. thesis, M.I.T. Available at `http://dspace.mit.edu/handle/1721.1/61549`.

[13] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A universal modular ACTOR formalism for artificial intelligence*. In: *Third Intl. Joint Conf. on Artificial Intelligence*, pp. 235–245.


## A Extended example: "File System"

In this section, we examine a "file system" actor based on a sketch given in the original SYNDICATE paper [9]. The responsibility of the "file system" is to store the contents of a named collection of *files*, and to provide some way of reading, writing and deleting such files.

The protocol accepted by the file system is simple:

```
(struct file (name content)) ;; Assertion: "The content of file 'name' is 'content'"
(struct save (file))         ;; Message: the contained file record should become current
(struct delete (name))       ;; Message: deletes the named file
```

There is no special message or assertion type needed to read files. Interested actors simply subscribe to `file` structures with the appropriate name and are notified as the file content changes.

The complete reactive file system implementation is shown in Figure 4. Let us examine its behaviour when run alongside an actor that monitors the contents of `novel.txt`:

```
(actor (forever #:collect [(files (hash))]
                (on (asserted (observe (file $name _)))
                    (until (retracted (observe (file name _)))
                           #:collect [(content (hash-ref files name #f))]
                           (assert (file name content))
                           (on (message (save (file name $content))) content)
                           (on (message (delete name)) #f)))
                (on (message (save (file $name $content))) (hash-set files name content))
                (on (message (delete $name)) (hash-remove files name))))
```

Figure 4: File System example: "Reactive" SYNDICATE Implementation

```
(actor (forever (on (asserted (file "novel.txt" $text))
                    (display "The novel's content is: ") (write text) (newline))))
```

When this novel-monitoring actor starts, it asserts its interest in the content of the novel, which causes an assertion `(observe (file "novel.txt" _))` to be placed in the dataspace. This in turn triggers the file system's `(on (asserted (observe (file $name _))) ...)` facet, which enters a state in which it asserts the current contents of the novel—initially `#f`, signifying a nonexistent file—and watches for updates.

If now some actor executes `(send! (save (file "novel.txt" "It was a dark and stormy night")))`, then *both* the file system actor itself *and* the `until` clause maintaining the `file` assertion will receive a message. The file system actor updates its private state—a hash table mapping file names to file contents—and the `until` clause updates its own state, which causes its `(assert (file name content))` facet to be re-evaluated. The novel-monitor in turn receives the update as a new assertion notification, and the change is displayed on the console for the user to see.

The novel-reader shown is simple-minded. A more sophisticated actor might become bored with its reading and retract its interest in the novel; the `until` clause in the file system would then terminate, retracting the assertion of the novel's contents in turn. The file-system actor thus not only provides a key-value store and a change-notification service, but also manages a kind of *cache*, reflecting certain values from the authoritative store (its `files` state variable) into the shared dataspace. The lifetime of each activation of the `until` clause scopes the lifetime of the corresponding cache entry and of the ongoing conversation, simple though it is in this case, with the monitoring actor.

An approximately equivalent "*non*-reactive" program is shown in Figure 5. The file-system actor has behaviour function `file-system-event-handler`, while each activation of the inner `until` clause is modelled as a separate actor with behaviour `file-observation-event-handler`. Missing is the linkage between the two classes of actor that is automatically supplied in the "reactive" case. Interesting points of difference include the manual repetition of patterns (e.g. lines 5 and 20, 13 and 21, and 15 and 22), managed automatically by the reactive constructs; the manual update of assertions in the shared space (lines 26–27), again managed automatically by the "assert" facet in the reactive implementation; the explicit use of projection and iteration over assertion sets (lines 5 and 32), managed automatically by the reactive `until` form; and, not least, the great difference in size. The "non-reactive" implementation is 38 lines long, while the reactive implementation is only 9 lines long.

```
(define (file-system-event-handler e files)
  (match-event e
    [(? patch? p)
     (transition files
                 (for-trie/list [((observe (file $name _)) (patch-added p))]
                   (define initial-content (hash-ref files name #f))
                   (spawn (file-observation-event-handler name)
                          initial-content
                          (patch-seq (assert (file name initial-content))
                                     (sub (observe (file name ?)))
                                     (sub (save (file name ?)))
                                     (sub (delete name))))))]
    [(message (save (file name new-content)))
     (transition (hash-set files name new-content) '())]
    [(message (delete name))
     (transition (hash-remove files name) '())]))

(spawn file-system-event-handler
       (hash)
       (patch-seq (sub (observe (file ? ?)))
                  (sub (save (file ? ?)))
                  (sub (delete ?))))

(define (update-file old-content name new-content)
  (transition new-content
              (patch-seq (retract (file name old-content))
                         (assert (file name new-content)))))

(define ((file-observation-event-handler name) e content)
  (match-event e
    [(? patch? p)
     (when (not (set-empty? (project-assertions (patch-removed p)
                                                (observe (file (?!) ?)))))
       (quit))]
    [(message (save (file (== name) new-content)))
     (update-file content name new-content)]
    [(message (delete (== name)))
     (update-file content name #f)]))
```

Figure 5: File System example: "Non-reactive" SYNDICATE Implementation